RunFein: a rapid prototyping framework for Feistel and SPN-based block ciphers

Khalid, A., Hassan, M., Paul, G., & Chattopadhyay, A. (2016). RunFein: a rapid prototyping framework for Feistel and SPN-based block ciphers. *Journal of Cryptographic Engineering*, *6*(4), 299-323. https://doi.org/10.1007/s13389-016-0116-7

Published in:
Journal of Cryptographic Engineering

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
© Springer-Verlag Berlin Heidelberg 2016.
This work is made available online in accordance with the publisher's policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Download date:04. May. 2023

Noname manuscript No.
(will be inserted by the editor)

RunFein: A rapid prototyping framework for Feistel and SPN based block ciphers

Ayesha Khalid · Muhammad Hassan · Goutam Paul · Anupam Chattopadhyay

Received: date / Accepted: date

Abstract Block ciphers are the most prominent symmetric-key cryptography kernels, serving as fundamental building blocks to many other cryptographic functions. This work presents RunFein, a tool for rapid prototyping of a major class of block ciphers, namely product ciphers (including Feistel network and Substitution Permutation Network (SPN) based block ciphers). RunFein accepts the algorithmic configuration of an existing/ new block cipher from the user through a GUI to generate a customized software implementation. The user may choose from various microarchitectural templates (unrolled, pipelined, subpipelined) to generate an HDL description of the cipher. Various modes of operation and NIST test suite may also be included. This high-level design approach eliminates the laborious development efforts for VLSI realizations of block ciphers. It enables a quick design exploration, consequently enabling fast benchmarking in terms of critical resource estimation of various versions / configurations of a cipher that vary in terms of security, complexity, performance. Using RunFein, we have successfully implemented some well known product ciphers and benchmarked their performance without significant degradation against their published literature.

Keywords Block cipher · Feistel network cipher · SPN cipher · Product cipher · High-level Synthesis · Rapid Prototyping · VLSI Implementation · Loop unrolling · Bitslicing · Subpipelining

A. Khalid, M. Hassan
Institute for Communication Technologies and Embedded Systems (ICE),
RWTH Aachen University, Aachen 52074, Germany.
E-mail: {ayesha.khalid, hassanm}@ice.rwth-aachen.de

G. Paul (*Corresponding author*)
Cryptology and Security Research Unit (CSRU),
R. C. Bose Centre for Cryptology and Security,
Indian Statistical Institute, Kolkata 700 108, India
E-mail: goutam.paul@isical.ac.in

A. Chattopadhyay
School of Computer Engineering,
Nanyang Technological University (NTU), Singapore.
E-mail: anupam@ntu.edu.sg

1 Introduction and Motivation

The world of cryptography is highly dynamic where newer cryptographic proposals, attacks are reported frequently. Competitions for inviting newer / better block ciphers, stream ciphers, hash functions get an active participation from an ever increasing cryptographic community. A thorough evaluation of these proposals on software/ hardware platforms follow. RunFein aids the cryptographer by enabling a high-level design approach for rapid prototyping of a block cipher *algorithmic* and *microarchitectural* specifications as software and hardware implementations. This section gives two major reasons for motivation of developing the need of a rapid prototyping framework for cryptographic functions, followed by methodology adopted by RunFein tool and our scientific contributions.

1.1 Cryptography is Dynamic

Below we discuss the major reasons fueling the ever-changing nature of cryptography. For each case, we give one example (out of many possible) to illustrate the point.

1. *Cryptanalysis*: Successful cryptanalytic attempts render the further use of attacked ciphers vulnerable as well as open doors for newer subsequent proposals. Countering cryptanalysis also often requires a modification in the original proposal, e.g., RC4$^+$ [11].
2. *Better machines*: Development of *Custom hardware* aids cryptanalytic attacks by enabling even the brute force attacks for small key sized proposals today, DES can today be broken in less than a day [10]. Moreover, *architectural updates* in computing machines influence cryptographic schemes, e.g., BLAKE [14], a hash function supports different word-size versions to cater both 32/64 bit machines.

This is an extended version of the conference paper [26] by Khalid, Hassan, Chattopadhyay and Paul, presented at ICISS 2013. Sections 1 and 3 are based on [26] with major revision and refinement. Sections 2, 4, 5, 6 and 7 are completely new contributions in this work.

Table 1 Parameters and their respective units to evaluate the performance of a cipher on H/W and S/W platforms

		Software Platforms			Hardware Platforms		
		μ controllers	GPPs	GPUs	ASICs	ASIPs	FPGAs
Security		key size, IV size, block size, number of rounds, mathematical soundness, known attacks					
Cost		The buying cost of the cipher IP license					
Performance	Throughput	cycles/block			blocks/second		
	Latency	N/A			cycles/block		
Resources	Core Area	N/A			NAND GE, mm^2		no. of LUTs
	Energy	Joules			Joules, Joules/access (RAMs)		
	Data	bytes	N/A	registers	bytes		
	Program	lines of code			N/A	bytes	N/A
Device occupancy		N/A		Cores usage%	N/A		LUT, RAMs usage%
Flexibility		N/A			key sizes, block sizes, modes, encryption/decryption etc.		

3. *Newer applications*: The imminent ubiquitous computing era has initiated newer security applications, e.g., *lightweight cryptography* for resource constrained devices. Consequently, lightweight cryptographic proposals aiming a thrifty area-power budget with reasonable security are frequently proposed, e.g., PRESENT [12].
4. *Design trade-off*: Most of the block cipher proposals support multiple modes of operation and versions for variable sized key, block size, rounds etc. These versions let the user choose a performance-security trade-off, e.g., varying the number of rounds in Salsa20 [15].

1.2 Evaluating a Cipher's Quality is Hard

From a cryptanalytic point of view, attack resilience is the most critical parameter for evaluating soundness of a cipher. For two ciphers with comparable estimated security level, based on their resistance to major cryptanalysis efforts over time, the mutual evaluation should however be based on their *performance*. The performance criteria depends on the application class; lightweight cryptographic functions support less complex functions compared to their conventional counterparts and hence require lower area, power but entertain smaller key sizes, lower throughput performances. These conflicting requirements of security and performance suggest considering interesting trade-off design points [7,6].

Evaluating a cipher's quality is hard since there are multiple versions/ modes of operations, evaluation parameters and implementation platforms to choose from. Table 1 gives a glimpse of multiple parameters used typically to evaluate the suitability of a cipher for a particular implementation platform. Developing custom computing architecture and mapping on known processors are termed here as the hardware and software implementation platforms, respectively. For software platforms, throughput of a cipher is specified in terms of cycles/byte (stream ciphers, PRNGs), cycles/hash (hash functions) or cycles/block (block ciphers). For hardware platforms, the *basic* parameters mentioned in Table 1 are sometimes taken up as *hybrid* combinations, e.g., Energy/bit, Throughput Per Area Ratio (TPAR), power-Area-Time (a triple product to quantify design compactness, throughput and power consumption), etc. Moreover, we may have multiple performance figures on the same computing platform according to the software optimizations or hardware configurations chosen for cipher implementation.

1.3 The RunFein Methodology

Considering the dynamic nature of cryptography, the in-pour of new ciphers requiring benchmarking against its existing counterparts is frequent. Various cryptographic competitions, including AES [38], NESSIE [39], CRYPTREC [40], eSTREAM [8], SHA-3 [41], CAESER [42], received an ever increasing number of candidate proposals compared to their successors. Performance is considered a critical criteria for filtering out the finalists out of these proposals. The call for AES [38] announced that the *computational efficiency* of both hardware and software implementations would be taken up as a decisive factor for selection of the winner. Similarly, the choice of Keccak as SHA-3 finalist was attributed by NIST to both its good software performance and excellent hardware performance [43]. Given the increasing importance of fair and fast hardware performance benchmarking, a rapid prototyping tool specific to cryptographic functions is but *imperative*.

Quantifying the performance of the cryptographic proposals as custom VLSI implementations requires benchmarking against diverse parameters like area, power, throughput, latency etc. The human-driven process of writing and validating HDL for stream ciphers is slow, error prone and requires expertise both in algorithm and hardware design domains to reach the options best suited for an application requirement. The workload is further compounded by the possibility of benchmarking various points on security-performance

trade-off by various microarchtitrues exploiting multiple levels of parallelism and bitslicing. RunFein aims to solve these problems through *automation*.

Various high-level synthesis (HLS) tools have been proposed both academically and commercially to automate the VLSI design cycle. Their architectural optimizations remain however *generic*. The user does not have the freedom to choose various hardware microarchitectures specific to cryptographic functions class to rapidly explore performance-resources trade-off. Some of these tools have slow learning curve as they require learning a new language. Moreover, the HDL generation performance shows a dependence on the coding style of the designer. Consequently their results remain suboptimal compared to the hand optimized cryptographic implementations. An HLS effort for SHA-3 candidates revealed that ranking of candidate algorithms in terms of performance remains the same independently whether the HDL implementations are developed manually or generated automatically using high level synthesis tool [44]. However, these implementations do not match the efficiency of manually written RTL, the frequency and throughput is lower and the area is up to 30% more compared to manual implementations [44].

RunFein presents a *language independent* interface; it accepts a sophisticated high-level block cipher design through a GUI. The user provides three sets of parameters to the toolflow. Firstly, the *algorithmic design configuration* comprising of constructive elements coming from a set of *functionally complete* constructive elements to define any block cipher. Secondly, the user chooses *microcarchtrual configuration* of the cipher for HDL generation. It includes a mode of operation and one of the various microarchitectures like unrolled, pipelined, subpipleined, bitsliced implementations. Thirdly and optionally, the user may specify a set of *testvectors*, if already known for the verification of the design. The software and hardware generation engines of the tool generate an optimized software implementation and a synthesizable HDL description. The design configuration given to the tool is validated for completeness and correctness at various stages of hardware/software generation. These rule checks detect functional and system-level problems much earlier in the design cycle improving design reliability and shortening time to market. The tool infers the necessary interfaces and structures to implement optimized HDL along with verification environments and necessary scripts. It provides a seamless end to end verification from the configuration to RTL validation/ verification environments.

1.4 Original Contribution

With a similar motivation as for RunFein, we earlier presented RAPID-FeinSPN [26], that caters to the rapid prototyping for block ciphers but covering only a simple loop folded hardware implementation. RunFein is a step forward in the direction of hardware optimizations by offering the user various microarchitectures design implementation alternatives. The noteworthy contributions of this work are listed.

1. We surveyed a diverse and wide range of block ciphers to systematically build up a *functionally complete* set of constructive elements/ architectural structures to define the configuration space of any block cipher.
2. RunFein allows a list of NIST standardized modes of operations in the software / hardware implementation of cipher.We integrated NIST test suite for evaluation of statistical randomness of the encrypted data.
3. For hardware implementation, the user has a choice to pick amongst various microarchitectures configured as per he wishes. This fast exploration of various design alternatives significantly shortens the cipher design cycle.
4. The configuration model completeness and RunFein tool effectiveness is validated by implementing some prominent block ciphers and benchmarking their performance to *rival* their manual implementations.

Rest of the paper is organized as follows. Section 2 discusses the categorization of block ciphers as computational kernels. Section 3 gives RunFein toolflow and discusses the configuration space of the product block ciphers. The salient features of the software generation engine of RunFein are discussed in Section 4. Section 5 gives the hardware microarchitectures supported by the hardware generation engine. Section 6 explains the area, power and throughput results of two prominent ciphers in various hardware configurations along with a comparison with existing work. Section 7 concludes this paper and provides future roadmap.

2 Dwarfs of Cryptography

For rapid prototyping of a block cipher, RunFein employs a *bottom-up design approach* by piecing together elementary operations to form a complete system. The idea is similar in spirit to the 13 computational kernel classes or so called *Berkeley dwarfs* capturing the major functionality and data movement pattern across an entire class of important application [27]. A similar idea is presented by Intel Recognition-Mining-Synthesis (RMS) view [28]. This concept of design

based on computational kernels has been exploited for rapid prototyping in cryptographic applications, e.g., fast hardware implementation of elliptic curve arithmetic operations [31], parameterized cryptanalytic toolflows [29,30], rapid prototyping frameworks for cryptographic protocols [32,33]. Undertaking these basic kernels across algorithms of an application class helps in a generic understanding as well as in an optimized implementation [34]. Classifying cryptography under computational dwarfs [27], makes it a subclass of *combinational logic* dwarf, along with other computing subclasses.

Next, we first justify why the study of block ciphers out of all the symmetric key cryptography functions are more significant and then investigate the computation kernels of block ciphers.

2.1 Workhorses of Symmetric Key Cryptography

Block ciphers enable *secrecy* of encrypted data, not beyond a single block of data. However under various *modes of operation* they enable data transmission having major services of Information Security (InfoSec) including *authenticity*, *integrity* and *confidentiality*. These modes transform block ciphers to other cryptographic primitives, making them the *workhorses* of symmetric key cryptography and consequently making their study imperative. Other than these operational modes, the basic deterministic transform functions of block cipher serve as elementary kernels or building blocks for many symmetric key cryptographic protocols. Fig. 1 highlights this *constructive* nature of the block ciphers being used as other cryptographic functions including stream ciphers, hash functions, message authentication codes (MAC) and cryptographically secure pseudo-random number generator (CSPRNG). We mention a few examples of cryptographic functions driven from block ciphers in this context.

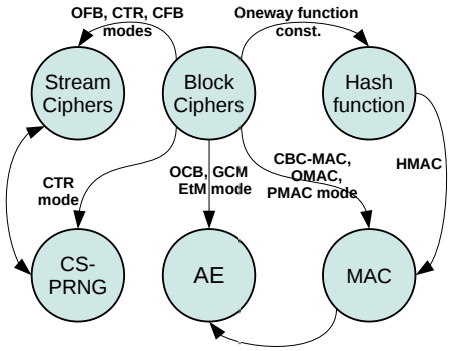

Fig. 1 Modes of operation for cryptographic functions

1. *Stream Ciphers*: Block ciphers are transformed to stream ciphers under counter mode (CTR) and output feedback mode (OFB) [16]. SOSEMANUK [17], an eSTREAM finalist stream cipher, uses a block cipher SERPENT for its construction.
2. *Hash functions*: Hash functions may be driven from a block cipher, operating in schemes that make them non-invertible one-way compression functions. WHIRLPOOL is based on an AES like block cipher operating under a Miyaguchi-Preneel hashing construction scheme [19]. More examples borrowing block cipher constructions include two SHA-3 finalists BLAKE [14] and Skein [18].
3. *MACs*: MACs may be driven from hash functions (in HMAC mode) or from block ciphers (in OMAC, PMAC and CBC-MAC mode).
4. *CSPRNG*: A CSPRNG can be driven from a block cipher operating in counter mode of operation. Also, running a stream cipher on a counter returns a CSPRNG, with its initial state kept secret.
5. *Authenticated Encryption*: Authenticated encryption is generically constructed by combining a block cipher and a MAC operating under a mode of operation, hence simultaneously providing confidentiality, integrity and authenticity assurances on the data. Various modes of authenticated encryption have been standardized by ISO [20].

Its worth highlighting that though block ciphers may serve as the building blocks of many cryptographic functions, these functions may have other roots of origin. Most of the popular stream ciphers are constructed using LFSRs along with some non-linear combining functions and an FSM. Similarly, many CSPRNGs originate from number theory problems. Also worth mentioning is the fact that cryptographic functions take inspiration from each other too. SEAL, HC-128 and HC-256 are stream ciphers that make use of SHA family of hash functions for their key expansion phase, SHACAL is a block cipher based on SHA-1. Many stream ciphers and CSPRNGs have common roots.

2.2 Ingredients of a Block Cipher

This section presents classification and typical elements of construction for block ciphers. Since our goal is to *define* configuration space of block ciphers for high level synthesis, we strictly focus on their architectural/ operational constructs. Their complexity and cryptanalytic properties are therefore skipped but could be referred from [45, Chapter 7].

A block cipher is a mapping of a *plaintext* data block of size S_B(blocksize) to an equal sized *ciphertext* block under the parametrization of a *key* (of size S_K, keysize). This deterministic mapping (*encryption*) should be invertible. The inverse function (*decryption*) generates the original plaintext given the ciphertext under

the *same* key. Classical/ historical block ciphers include Caesar ciphers, affine ciphers, substitutions ciphers, polyalphabetic substitutions, etc. These techniques are proven over time to be cryptanalytically vulnerable and not suitable for practical use today [45, Chapter 7].

The product ciphers make the most popular class of block ciphers (and lightweight block ciphers) used today. A *product cipher* combines multiple data transformations so as to make the resulting cipher is more secure than the individual transformations. These *transformations* may include permutations (adding diffusion), substitutions (adding confusion), translations (e.g., XOR), linear transformations (e.g., rotation), arithmetic operations, modular multiplication, transpositions etc. An *iterated product cipher* involves sequential repetition of a set of transformations called a *round* function. The round function iterate N_r (round-count) number of times during encryption/ decryption. For i^{th} round a $subkey_i$ (of size S_{SK}) is generated. Two major classes of iterated product ciphers are defined as follows [45, Chapter 7].

1. *A substitution-permutation (SPN)* cipher is an iterated product cipher composed of a number of stages each involving substitutions and permutations. To ensure inevitability, for each of the generated subkeys the round function is a bijection on the round input.
2. *A Feistel cipher* is an iterated product cipher whose each round *splits* data, passes one half to the round function and *swaps* the two data halves. Hence Feistel cipher operate on alternating halves of the ciphertext, while the other remains constant. The round function need not be invertible to allow inversion/decryption of the Feistel cipher.

2.3 Computational Building Blocks of Symmetric Key Cryptography

This section attempts to unconventionally classify the major functions of symmetric key cryptography (block ciphers, stream cipher, hash functions) based on their underlying common computational elements.

A small set of three operations, i.e., modular addition (A), bit rotation (R) and bit wise XORing (X) make a *functionally complete* set of operations for building any cryptographic function [24, Section 5], including block ciphers, stream ciphers and hash functions. The term AXR (later renamed to ARX) was coined by Ralf-Philipp Weinmann [21] in 2009, however such designs have been proposed much earlier. This combination of linear (X, R) and nonlinear (A) operations, iterated over multiple rounds achieves strong resistance against known cryptanalysis techniques [24].

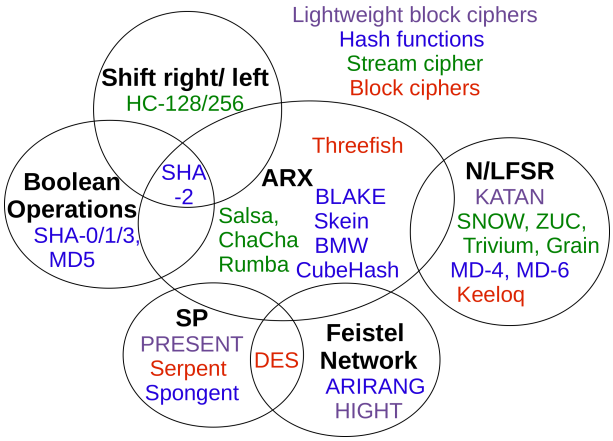

Fig. 2 Computational commonalities of symmetric key cryptographic functions

Fig. 2, a subset diagram, captures the computational kernels of symmetric key cryptography. The bitwise shift operation is added to the ARX pool of operations for construction of some new cryptographic functions like HC series of stream ciphers [5]. Similarly, addition of Boolean operations (AND, NOT) make the computational basis of hash functions including MD5 [23] SHA-0,1 [3] and SHA-3 [22]. Interestingly, SHA-3 (Keccak) originates from the concept of flexible *sponge constructions* for cryptographic functions, however, classification based on the underlying operations brings SHA-0,1 and SHA-3 simply under the common axis. SHA-2 [3] requires bitwise shifting as well as Boolean operations in addition to the ARX pool of operations, indicated in Fig. 2.

Many stream ciphers are based on a Linear/nonlinear Feedback Shift Register (N/LFSR) whose inputs are selected from the previous state after linear/nonlinear functions applied on them. Taking examples from eSTREAM finalists [8] include SOSEMANUK, and all its three finalists in the hardware profile. FSRs have been employed in the construction of block ciphers and hash functions too, some examples are listed in Fig. 2.

Feistel ciphers may use substitutions and permutations, other than the ARX operations, in their round functions. XOR is generally used for key whitening the round values with subkey of that round. Addition operation might not be explicitly used in round operations, however, a count-up/down counter is always required for encryption/decryption block realization, respectively. DES has a Feistel structure but employs SBoxes and PBoxes for its round operation. AES [2]

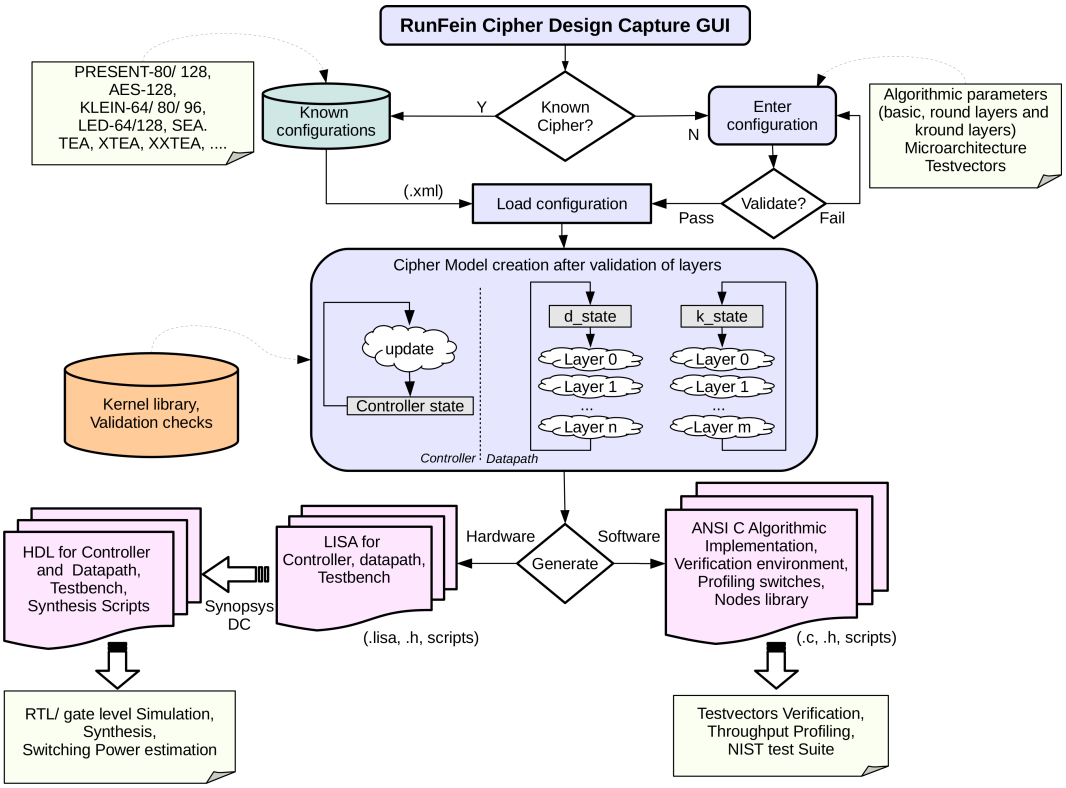

Fig. 3 RunFein toolflow for software generation, LISA based hardware generation and NIST test suite

does not have any PBoxes and rather uses Galois field multiplication. Its noteworthy that this computational categorization highlights only the commonalities as a trend in cryptographic functions. This categorization is neither complete nor by definition binding to a particular class of ciphers. Consequently, exceptions exist, e.g., TEA [4] family of lightweight block ciphers (XTEA, XXTEA) are Feistel Network ciphers by structure and use shift operations other than ARX. AURORA, a hash cipher for SHA-3 competition has a structure as a combination of SPN and a generalized Feistel structure [25].

Classifying the cryptographic functions on the basis of their primitive computational elements brings forward a *surprisingly simplistic* angle of viewing them, beneficial to their implementation, both on hardware and software platforms. RunFein is developed around the concepts of *modularity* and *extensibility*. It supports constructive composition of cryptographic building blocks supporting SPN / Feistel network based block ciphers, which are favorite primitives for block ciphers today. Additionally, stream ciphers based on block ciphers (e.g., salsa20 [15]) can be realized using RunFein. It can also model Lai-Massey structure block ciphers and can be conveniently extended to support newer structures/ components if/ when the need arises.

3 RunFein Toolflow

The toolflow of RunFein is graphically shown in Fig. 3. The user populates the configuration space of a new block cipher to get customized software and hardware implementations. A sophisticated design capture is made possible via a GUI to let the user conveniently specify cipher design and implementation customization. The configuration for a cipher could be added, parameter by parameter, or could be saved and loaded later. A list of known ciphers is available to instantly load the configurations for easier manipulation. RunFein validates this design capture for completeness and correctness at various stages of the toolflow. It successfully abstracts away the diversity of the design space by translating the configuration it to a generic block cipher template. The configurations undergo a set of design rule checks before generating the software and hardware implementations.

3.1 Cipher Configuration Space

A key challenge addressed in this work is to identify a *complete* set of algorithmic primitives and architectural sub-structures that is generic enough to configure a range of block ciphers and their implementations. After survey of diverse ciphers, we developed sub-structures and component lists to develop primitive libraries for

Table 2 RunFein supported modes of operation for block ciphers

Mode of operation	Encryption	||	Decryption	||	Initialization Vector
Electronic codebook (ECB)	$C_i = E_k(P_i)$	✓	$P_i = D_k(C_i)$	✓	-
Cipher block chaining (CBC)	$C_i = E_k(P_i \oplus C_{i-1})$	✗	$P_i = D_k(C_i) \oplus C_{i-1}$	✓	$C_0 = IV$
Propagating CBC (PCBC)	$C_i = E_k(P_i \oplus P_{i-1} \oplus C_{i-1})$	✗	$P_i = D_k(C_i) \oplus P_{i-1} \oplus C_{i-1}$	✗	$P_0 \oplus C_0 = IV$
Cipher Feedback (CFB)	$C_i = E_k(C_{i-1}) \oplus P_i$	✗	$P_i = D_k(C_{i-1}) \oplus C_i$	✓	$C_0 = IV$
Output Feedback (OFB)	$C_i = P_i \oplus O_i$	✗	$P_i = C_i \oplus O_i$	✗	$O_i = E_k(O_{i-1}), I_0 = IV$
Counter (CTR)	$C_i = P_i \oplus E_k(IV)$	✓	$P_i = C_i \oplus D_k(IV)$	✓	$nonce + counter = IV$

software and hardware realizations. We propose a so-called *layered architecture* where each layer specifies a data transformation specified by *operation*. To fully appreciate the concept of layers of operations, we consider the data flow graph of the cipher (and its key expansion) where data is moving from top to bottom. The layers are then the *horizontal divisions* of the data flow diagram.

The configuration parameter set is categorized into *algorithmic parameters*, *modes of operation*, *microarchitectural parameters* and *testvectors*. (All parameterizable attributes that a user must populate are highlighted in the proceeding discussion).

3.1.1 Algorithmic Parameters

The parameters to define the algorithmic construction of block cipher are

- *Basic Parameters:* The input **plaintext** to a block cipher (encryption) and its output **ciphertext** are of equal size, blocksize $\mathbf{S_B}$ (all sizes specified in bits). The size of **Key** is specified as $\mathbf{S_K}$ (for some operational modes an **IV** (initialization vector) having size $\mathbf{S_{IV}}$ must also be specified). The granularity of the cipher is specified as wordsize ($\mathbf{S_W}$) of the cipher. Block ciphers iterate a deterministic combination of operations known as a **round**. The rounds operate $\mathbf{N_r}$ (roundcount) number of times during encryption/ decryption.
- *Round Layers:* For most block ciphers, the data undergoes an initial and/or final transformation, before and/or after the rounds processing, respectively. Since these transformations may differ from each other and the central round transformation, we name them as **round_init** and **round_final**, while the round transformation is referred as as **round_middle**. These three kinds of rounds are defined by a series of **layers** of operations. Every layer comprises of at least one of the following **operations**, performed exclusively on the user specified portions of the layer input (this list could be conveniently extended to accommodate newer operations).
  1. Substitution or permutation boxes (**SBox**, **PBox**).
  2. Galois field multiplication **GF-mul** with another polynomial (primitive polynomial must be specified).
  3. Bitwise operations including **rotation**, **shifting**, **addition**, **XOR-ing**, **ARK** (add round key), AddCounter (XORing with counter).
  4. Operations specific to Feistel networks including **split**, **swap**.
  5. Compund operations used in popular ciphers, e.g., **shiftrows**, MixColumns used in AES [2]
  6. No operation (**nop**)
- *Kround Operation:* For each *round*, a **subkey** (of size $\mathbf{S_{SK}}$) is generated through key expansion. Like rounds, key expansion requires iteration of **kround** transformation, N_r (roundcount) number of times to generate subkeys. *kround* may also have different definitions for **kround_init**, **kround_middle** and **kround_final**, each of them are defined by *layers* of operations like cipher *rounds*.

The *layers* of each round have a **layernumber** to specify their order of execution, within that round. The input to and output from a layer may differ in size (bits) due to an expansion/ contracting layer operation and is specified as **S_lin** and **S_lout**, respectively.

3.1.2 Modes of Operation

RunFein lets the user opt from a list of modes of operation to add the chaining dependencies between adjacent blocks of data during encryption / decryption. Currently, any of the NIST standardized modes of operation may be chosen for implementation [16], as listed in Table 2. Here C_i represents ciphertext for the i^{th} plaintext block after encryption function parameterized by the secret key E_k while P_i represents the plaintext after decryption. Due to the chaining dependencies, multiple blocks of data cannot be subjected to encryption or decryption in a parallel fashion for some modes of operations as indicated in the Table 2. For all modes other than ECB, the user specifies the IV and any additional parameters required.

3.2 Putting the Things Together

We take up two ciphers and try to workout their algorithmic configuration according to the discussed RunFein's layered architecture definition methodology. These being AES-128 [2] due to its widespread usage and PRESENT [12] due to its ultra lightweight nature. Moreover, both of these ciphers have been standardized by ISO. We define the configuration space for these ciphers in encryption blocks only. The reader is kindly requested to refer to the documentation of these ciphers for a detailed understanding of their functionality [12, 2].

Table 3 RunFein basic parameter configuration space

Parameter	PRESENT-80 [12]	AES-128 [2]
S_B(bits)	64	128
S_K(bits)	80	128
S_{SK}(bits)	64	128
S_W(bits)	4	8
N_r(rounds)	32	10
round_init	-	1 layer
round_middle	3 layers	4 layers
round_final	1 layer	3 layer
kround_init	-	1 layer
kround_middle	3 layers	7 layers
kround_final	-	-

3.2.1 PRESENT-80

Table 3 shows the basic parameter configuration space for 80 bit key of PRESENT cipher. The configuration parameters for PRESENT-80 are fed to the tool's GUI and are stored as an *xml* configuration file, a snapshot of which is shown in Fig. 4. A separate token ALGORITHM holds the basic parameter, round and key round operational layers. Basic parameters includes sizes of block, key and word size and the information of rounds for encryption. The ROUND token holds the information for three types of rounds. A *round_init* is not required hence no layers are defined for it. The *round_middle* is defined by the following 3 *layers* of operation:

- *layer*0 is the ARK, where input and output to the layer is equal sized. Data is xored with the subkey, where
  $subkey = key[79 : 16]$.
- *layer*1 is the $SBox$, the user specifies a total of 16 SBoxes ($\frac{S_B}{S_W}$) to be inserted, specified by *Word2Sub* being '-1'. For SBox 2^{S_W} values are specified, each S_W bits wide.
- *layer*2 is the $PBox$, with a total of S_B arguments $\in [0..S_B]$.

The arguments for SBox and PBox can be loaded either by a text file or added by the user in the edit boxes. The *round_final* is specified by one layer of ARK, same as the first layer of *round_middle*. Hence the ciphertext is taken out after the first ARK layer in the last iteration of cipher encryption. Fig. 4 shows the KROUND token that configures the key expansion information. For key expansion in PRESENT-80, the *kround_init* and *kround_final* are not required and hence defined as having no layers. The *kround_middle* requires three layers of operations defined below.

- *layer*0 is the $ROTATE$ operation configured to carry out a *left* rotation by *61*.
- *layer*1 is the $SBox$. The user specifies one SBox inserted at word number 19 of the key, the most significant nibble to the layer input. The rest of the bits are passed on un-altered.
- *layer*2 is the $AddCounter$ that XORs the selected bits of the data (bit 19 till 15) input to the layer with a 5-bit *counter* (round counter).

A round *counter* increments till it reaches $N_r - 1$ and a valid *ciphertext* is available.

3.2.2 AES-128 [2]

For AES-128, the corresponding parameters for RunFein are specified as given in Table 3. The *round_init* requires one operation layer, i.e., ARK. *round_middle* is defined by 4 layers.

- *layer*0 is $SBox$. The user specifies 16 SBoxes to be inserted along with SBox definition of 256 bytes.
- *layer*1 is a $Shift\text{-}rows$ operation. Its a compound operation that takes up the layer input as a 2-D matrix and and re-arranges the words of each rows with fixed offsets.
- *layer*2 is a $GF - Mix$, a compound operation assuming 2-D arranged data. The user specifies a 4x4 column coefficients for GF(2^8) multiplication.
- *layer*3 is the ARK, that XORs the key with the data.

Using this layered architecture, a cipher may have multiple valid definitions. The $Shift\text{-}rows$ operation in *layer*1 may have been defined using various layers, each rotating one row of the state matrix, as defined by the AES specifications. We define it as a standard *compound operation* since it's a common operation used in ciphers other than AES, e.g., LED.

The *round_final* is defined by 3 layers, same as *layer*0, *layer*1 and *layer*3 of *round_middle*. For each *round* a *subkey* is generated through a *kround*. The *kround_init* is a *nop* layer since the first subkey is the

Fig. 4 The configuration file snapshot for PRESENT-80 generated by RunFein

input key itself. The *kround_final* is not required and hence not defined. *kround_middle* requires 7 layers of operations for its definition as shown in Fig. 5.

- *layer*0 is a *ROTATE left* by *8* layer. It takes the least significant 32-bit word of the key. This layer also expands 128 bits of input to 160 bits of output by concatenating the input bits unaltered along with the rotated word output.
- *layer*1 is the *SBox*, 4 SBoxes are inserted on the 4 least significant bytes of layer1 input.
- *layer*2 is a *XOR* with *counter* dependent constants (RCON). The constants are specified by the user using a text file.
- *layer*3-*layer*6 are *XOR* operations, performing selective xoring of layer inputs as per AES specifications.

Fig. 12 shows a GUI snapshot of RunFein with AES-128 configuration.

3.3 Cipher Model Creation and Validation

The RunFein framework provides a sophisticated configuration capture via a GUI (Some snapshopts of the RunFein GUI are presented in the appendix). It provides convenient default values in the GUI wherever necessary, the configuration file with default values for *microarchitecture* and *testvectors* for PRSENT-80 is shown in Fig. 4 (further discussion follows in following section). Other than the parameters, specified by the user through GUI, some parameters are *inferred* by the tool. A **counter** is required to keep track of the iterations of the cipher. It counts up or down during encryption or decryption of a block of data, respectively. Its size is taken up as $ceil(\log_2(N_r))$ bits. Other than counter, we have two variables namely **d_state** and **k_state**, that contain the updated data state and key state, respectively (for hardware implementation these values are D-flipflops instead).

Before creation of a valid cipher model, the configuration parameters given by the user undergo a list of defined rules checks. The user is prompted in case of a violation and cipher implementation does not proceed unless a valid configuration is specified. (Some additional rules related to hardware microarchitectures are discussed in Section 5.3.6.)

- Blocksize of any cipher by definition equals the sizes of plaintext/ ciphertext.
  $S_B = S_P = S_C$
  $S_B = 2m$, where $m \geq 1$.
- Size rules for key and the subkeys generated.
  $S_B = S_{SK}$
  $S_K = 2m$, where $m \geq 1$.
- Size rules for wordsize of cipher.
  $S_W = 2m$, where $m \geq 1$.
  $S_W \leq S_K$ and $S_W \leq S_B$.
  $S_W = n.S_B = k.S_{SK}$, where $n, k \geq 1$.
- For modes of operations (defined in Section 3.1.2), IV size rule
  $S_B = S_{IV}$
- The number of subkeys generated should be equal to the number of ARK operations, hence each ARK consumes one key.
- The SBox values are $\in [0..2^{S_W}]$.
- PBox, rotation/shifting, XOR operations have arguments $\in [0..S_B]$.
- The polynomial coefficients for GF-mul are not $\emptyset$.

The configuration file is parsed by RunFein and *cipher model* is created, for PRESENT-80 and AES-128 it is shown in Fig. 5. The cipher model comprises of a *controller* and *datapath*. The controller is simply the inferred *counter* (not shown in Fig. 5), the datapath of the cipher is constructed by operational layers of round and kround. A mulitplexer is also inferred at the input to *d_state* and *k_state* registers, controlled by the round count. For PRESENT-80, the last round or $round\_final$ comprises of ARK layer only and hence the cipher text is extracted after layer0. For AES-128, layer0 of kround expands the key and layer3 contracts it back to 128 bits again.

4 Software Generation Engine

The software generation engine takes either the user specified configuration of a new cipher or alternatively loads the design configuration of a known cipher ((Fig. 3)). One also specifies data for plaintext, key, IV (through text files or edit boxes) using the GUI. RunFein compiles the cipher model to generate a high performance, fixed-point ANSI-C description. The code is enhanced by a simulation environment with user controllable switches for verification, throughput profiling, data dumping etc. The generated code is not specifically optimized for a particular General Purpose Processor (GPP), however, it has a regular structure and good code readability.

All the configuration parameters of the cipher (as specified in xml file listing in Fig. 4) are *#defined* in a *header* file. This includes all basic configuration, test vectors and the microarhitachture, though software implementation only caters the default values of a simple iterative loop folded implementation. Data types of registers, layers and all interfaces are *typedef*-ed in accordance with their respective granularity specified. Supplementary *functions* are kept in a separate file, that is *included* in the main file during simulation. These functions include datatype conversion functions (e.g., conversion of hexadecimal to binary arrays and vice versa), data dumping and verbose simulations. For each operational layer of *round* and *kround*, a separate function is defined with interface and functionality, as per the user specified. Layers may operate on operands with different granularity, i.e., PBox operates on bits, SBox operates on S_W etc. The functions generated include relevant calls to conversion of granularity functions in addition to the functionality of the layer operation.

The main body of code, having the *controller* and the *datapath* of the cipher model, is a separate file that *#includes* all supplementary and header files. For elaboration of code simulation environment, we refer to the simplistic pseudocode for encryption of one block of data given in algorithm 1. The plaintext and key are assigned to the local variables *d_state* and *k_state*, respectively (line 1,2). *k_state* is updated first by $Kround\_init$ function. Using the updated key, the *d_state* is updated using $round\_init$ function(line 4). The controller part of the cipher comprises of *counter* variable, keeping track of the round under execution. The loop starting in line 6 iterates for $roundcount - 1$ times and keeps updating the data and key registers. The final round generates the last *k_state* which is used up by $round\_final$ to generate the ciphertext, as given in (line 9,10), respectively. RunFein generated code for AES is presented in the appendix of [26].

The software generation engine of RunFein generates a single-threaded, untimed, sequential C model of the stream cipher with necessary libraries and scripts. Some of its additional features are highlighted.

- **NIST Test Suite**: RunFein has the NIST test suite [1] integrated with it to characterize the statistical qualities of PRNGs. It serves as a first step in determining the suitability of a PRNG used for cryptographic purposes. Fig. 13 gives a GUI snapshot of RunFein for the selection and parameterization of various statistical tests available for execution as per the user

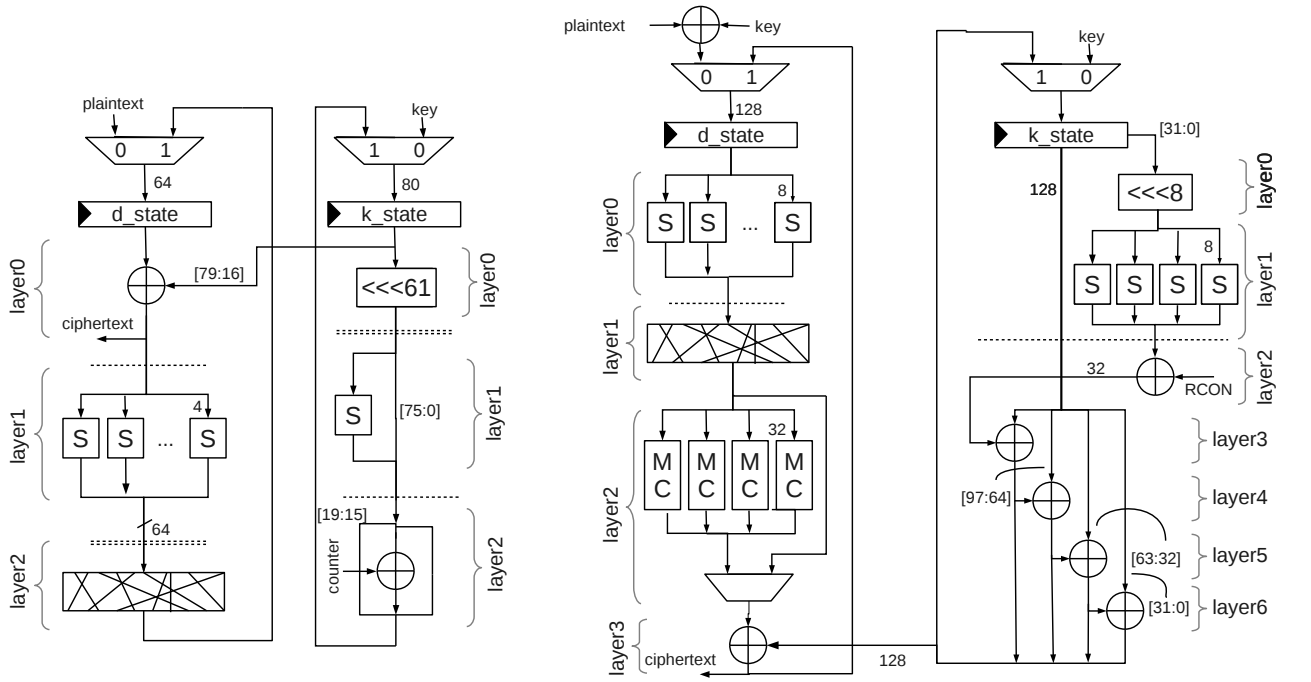

Fig. 5 Layers for the loop folded implementation (with on the fly key expansion) of PRESENT-80 (left) and AES-128 (right)

Input: $plaintext$, key, $configuration$
Output: $ciphertext$
1 $d\_state = plaintext$;
2 $k\_state = key$;
3 $counter = 0$;
4 $k\_state =$ **Kround_init(**$counter, k\_state$**)**;
5 $d\_state =$ **round_init(**$counter, k\_state, d\_state$**)**;
6 **for** $counter{=}1$ **till** $\leq N_r$ **step** 1 **do**
7 $\quad k\_state =$ **Kround_middle(**$counter, k\_state$**)**;
8 $\quad d\_state =$ **round_middle(**$counter, k\_state, d\_state$**)**;
end
9 $k\_state =$ **Kround_final(**$counter, k\_state$**)**;
10 $ciphertext =$ **round_final(**$counter, k\_state, d\_state$**)**;

Algorithm 1: RunFein Encryption Pseudocode

wishes. (RunFein caters only the block ciphers, however, they behave like stream ciphers and CSPRNGs under certain modes of operation.)

– **Verification**: For the verification of the generated model according to the user specified testvectors, a verification environment is generated. For new proposals, without defined testvectors, the verification switches may be turned off by the user.
– **Performance Profiling**: The user may enable a performance profiling environment in the generated software implementation to evaluate encryption speed (in seconds, cycles/ byte) of the cipher design. Provision of encrypting bulk data from random plaintext for monitoring data randomness is provided. A reasonably efficient generated implementation may be further manually optimized for a specific platform.

5 Hardware Generation Engine

The hardware generation engine requires additionally the microarchitectural configuration of the cipher model to be specified by the user, other than the algorithmic configuration to generate a complete working model of the block cipher in synthesizable *HDL* along with a testbench and necessary scripts. First the viability of the chosen microarchitecture configuration is evaluated by RunFein by a list of rule checks. After design validation, RunFein generates the design implementation as an ADL and relies on Synopsys Processor Designer [36] for generation of synthesizable HDL code, as shown in Fig. 3. This design can be profiled to get critical parameters like the maximum clock frequency of the design, chip area and power consumption.

5.1 HDL Toolflow

RunFein employs *Synopsys Processor Designer* for an efficient high-level synthesis framework [36]. The HDL design is generated in a high level language called *Language for Instruction-Set Architectures* (LISA) [35]. The language offers rich programming primitives to capture an implementation of a design with full programmability to an Application Specific IC (ASIC). The hardware implementation flow using LISA is shown in Fig. 6. Besides generating a complete set of software development tools (compiler, simulator, assembler, linker), *synthesizable HDL* code (both VHDL and

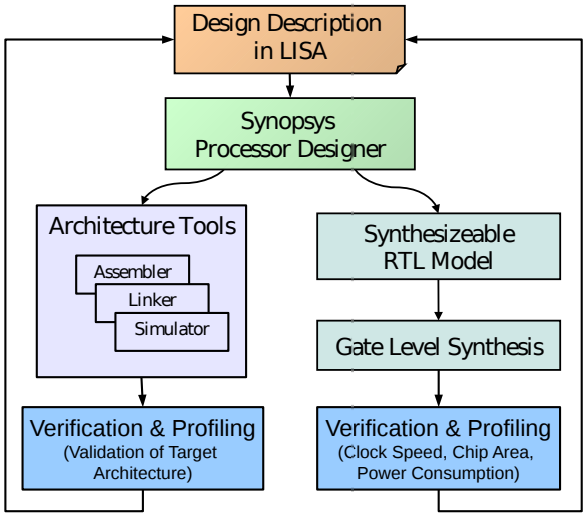

Fig. 6 Implementation flow with LISA

Verilog) for the design can be generated automatically from the LISA processor description. The language allows full control over minute design decisions and preserves the overall structural organization neatly in the generated hardware description.

The RunFein generated LISA description is converted to a synthesizeable, hierarchical block cipher HDL and testbench with necessary scripts that can be further used to carryout

- Simulations for design verification, gate-level simulation (post-synthesis) using verification tools.
- Logic synthesis of the design for profiling critical parameters like the maximum clock frequency, chip area.
- Post-synthesis power consumption estimation with using back-annotation.

5.2 Operational Layers Inference

The hardware generation engine performs tries to optimize the hardware reuse for middle and final rounds of the algorithm by gauging the commonalities between the two. Since for PRESENT-80, the *round_final* is a single ARK operation, the final ciphertext is therefore taken out after first layer of *round_middle*. For AES-128, the middle round and last round differ only in one layer, i.e., GF-mul. A bypass mux is automatically inserted, enabled at the final round as shown in Fig. 5. SBox sharing for bitsliced configurations is also performed (discussed in subsequent sections). If for a cipher, the ARK operation is in $round_int$ (e.g., in PRESENT-80) subkey is taken from the *k_state* register, if it is in $round_final$ (e.g., in AES-128), it is taken from the output of the last layer of *k_round*.

A simplistic mapping of the layers into operations is carried out. (The extensible RunFein framework enables/ encourages multiple customized LISA definitions of operations)

- SBoxes are implemented as *read-only lookup tables* (LUT).
- Diffusion operations like rotation, shifting, PBoxes are implemented using *rewiring* of the inputs and renders no overhead to the combinational delay of the circuit.
- GF-mul is implemented by shifting and XORing operations in accordance to the primitive polynomial of the finite field specified.
- Supported popular *compound* operations (e.g., MixColumns) have cascaded implementations of their constructive operations.

5.3 Supported Microarchitectures

Through RunFein the user can quickly explore various microarchitecture design options residing at different *intensity* of the performance-area trade-off. The user specifies algorithmic configuration of the cipher design always according to the simplistic loop folded architechture. In addition, he must specify the microarchitecture he wants RunFein to automatically implement. By tweaking the microarchitecture configuration, he may opt for *parallel* implementations (loop subpipelining/ unrolling) duplicating hardware for boasting throughput or *bitsliced* designs economizing area/ power at the expense of lower throughput performance by employing resource sharing. We discuss these microarchitectures individually, they are depicted in Fig. 7.

5.3.1 Loop Folded

A typical loop folded block cipher implementation performing one round per clock cycle (N_r cycles per block) is shown in Fig. 5 and Fig. 7 a). It is the default hardware implementation microarchitecture of RunFein and serves as a middle point for area-throughput trade-off between *parallel* implementations and *bitsliced* implementations. The controller comprises of round *counter* register, incrementing every cycle (Fig. 8 a). The selection of plaintext or folded data for *d_state* register is controlled by this register. A valid ciphertext is generated when counter register hits N_r.

5.3.2 Loop Unrolled

The loop unrolled configuration replicates round (and kround) resources u times to execute multiple rounds

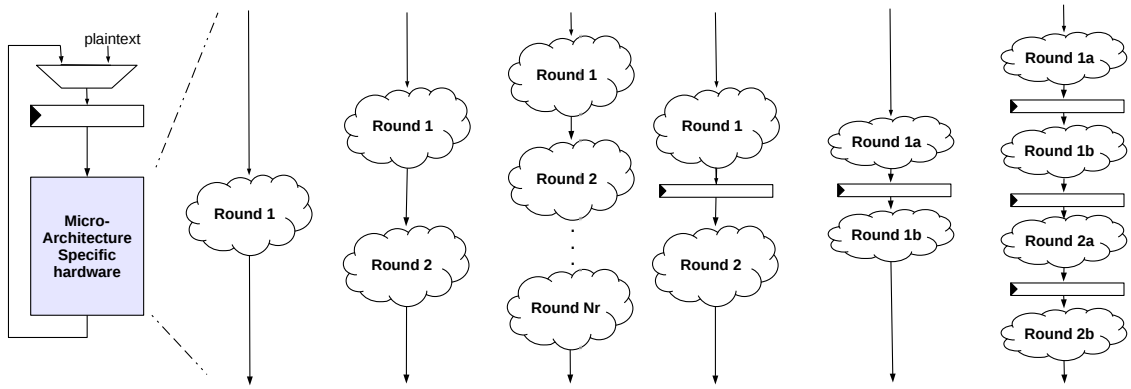

Fig. 7 Various parallel microarchitecture implementations supported by RunFein a) Loop folded b) Unrolled by 2 c) Fully unrolled d) unrolled by 2 with pipeline e) Subpipelined once f) Subpipelined once and unrolled by 2 with pipeline

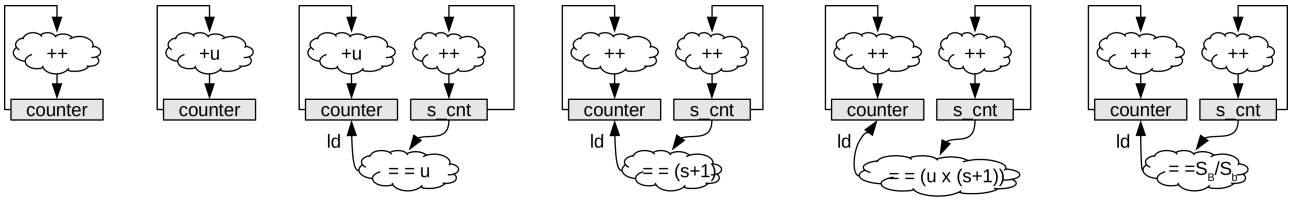

Fig. 8 Controller for Various microarchitecture implementations supported by RunFein a) Loop folded b) Unrolled by u c) Unrolled by u with pipeline d) Subpipelined by s e) Subpipelined by s and unrolled by u with pipeline f) Bitsliced with S_b

in one clock cycle, where u is the unrolling factor. Consequently the critical path of the circuit increases, decreasing the maximum operational frequency, the area also increases. The *counter* increments by u per cycle since the design require N_r/u cycles for encryption of a complete block (N_r/u not being a fraction), as shown in Fig. 8 b). A higher throughput performance is expected since the *propagation delay* and the *register setup time* come only once in the combinational delay for u rounds. This gain in throughput is hard to enumerate without experimentation hence synthesis profiling is required (a twice unrolled hardware configuration is shown in Fig. 7 b). Two critical design points relevant to the loop unrolling are

- A *fully unrolled* architecture with $u = N_r$ encrypts/ decrypts of data in a single cycle (Fig. 7 c). The RunFein hardware generation engine optimizes the hardware for the $round\_final$ if it is different from the $round\_middle$. The $round\_middle$ hardware is replicated $(u-1)$-times following the hardware for $round\_final$ instantiated once.
- A *loop unrolling with pipelining* architecture can be chosen by the user to automatically insert pipeline registers between unrolled rounds. Consequently, the critical path of the design also does not increase due to unrolling, this design handles *multiple* blocks of data simultaneously. Configuration in Fig. 7 d) processes two blocks of data in a total of N_r cycles boasting throughput by u. A supplementary counter or *s_cnt* keeps track of the unroll factor, which when fulfilled generates the load signal for counter to increment directly by u (Fig. 8 c). Hence in subsequent cycles, u-many valid ciphertexts are generated when counter equals the roundcount.

5.3.3 Subpipelining

Using RunFein the user may choose to insert a subpipeline between any two layers in a *round* to reduce the critical path of the design. To ensure data consistency, for s subpipelines inserted in a cipher *round*, an equal number of subpipelines should be specified by the user to be inserted in *kround* as well. To do so, the user must *check* subpipelining option *on* to be able to insert various operations along with a subpipeline register as shown in the GUI snapshop (Fig. 14). Insertion of each subpipeline increments the number of *multiple* blocks being processed, i.e., s subpipelines make the cipher design handle $(s+1)$ data blocks simultaneously (for $s = 1$ Fig. 7 e). A supplementary register *s_cnt* inserted keeps track of the subpipeline (Fig. 8 d). If the user wishes to insert a subpipeline within a layer, he must first redefine that layer as two layers, split at the cut-set point.

5.3.4 Hybrid Microarchitectures

Using RunFein the user may opt for some *hybrid* parallel microarchitecture configurations supporting both subpipelining and unrolling. Fig. 7 f) shows a hybrid

microarchitecture with subpipeline ($s = 1$) and unrolling with pipeline by a factor ($u = 2$). Its a multiple block configuration, handling 4 data blocks simultaneously. Consequently, the controller needs a supplementary register *s_cnt* to keep track of the total iteration count (Fig. 8 e).

5.3.5 Bitslicing

Through bitslicing, RunFein *tiles* the parallel loop folded architecture to work on S_b bits at a time ($S_b < S_B$). Consequently the design has lower area and lower throughput, a technique especially interesting for lightweight block ciphers. In most of the SPN ciphers, SBoxes account for a significant area portion, e.g., more than 30% of the PRESENT-80 loop folded implementation area is contributed by its 17 SBoxes [12]. Hence, S_b is generally taken as S_W or a multiple of it.

The krounds and rounds are sliced to operate the task of one cycle in S_B/S_b cycles. The controller of the bitsliced architecture changes so that the *counter* increments once after the *s_cnt* hits S_B/S_b. The encryption of one block requires $S_b \times N_r$ cycles as shown in the Fig. 8 f. The *d_state* and *k_state* are shift registers (with parallel load/ stores possible), with shift granularity of S_b. Hence the operations of each layer in a round is performed on S_b bits and the result is stored in *d_state* shift register. Similar to the bit slicing of S-boxes, operations like XOR, Addidion (with carry bit) can be bit sliced. However, for some operations, the operation slicing requires large extra selection logic, e.g., PBoxes, rotation. Since these bit manipulation operations (when performing in parallel configurations) have no logic overhead, its wiser not to bitslice them.

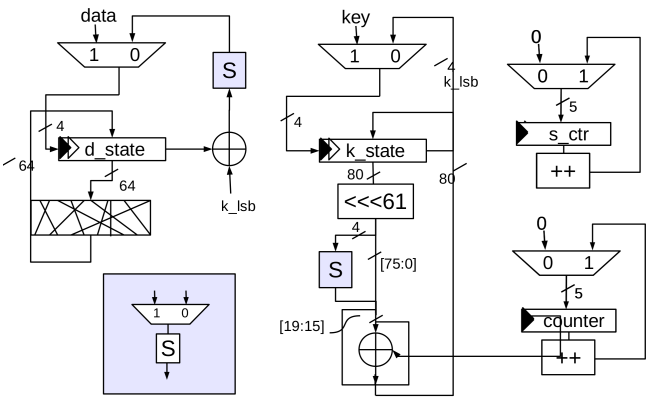

Fig. 9 Bitsliced implementation of PRESENT-80

RunFein takes the bislice factor (S_b) of a cipher and after evaluation the validity of the design generates a bit-sliced implementation. Fig. 9 shows a bitsliced $S_b = 4$ PRESENT-80 implementation requiring S_b/S_W (1) SBox per round, shared between *k_round* and *round* calculations. A similar design has been presented for smallest area footprint of PRESENT-80 in [13]. Since the Key expansion is generally non-expensive in terms of resources, bit slicing is not applied to *krounds*. Hence the key is loaded in S_K/S_b cycles in *k_state* shift register but a subkey is calculated in a single cycle. For the round calculation, 4 bits are xored with one key nibble and passed through the SBox in each cycle. As PBox is not bit sliced round calculation requires S_B/S_b cycles plus one for PBox calculation. Since the key expansion requires only one SBox, the round and kround share one. Through RunFein, the bisliced and optimized designs of ciphers having compact HDL implementation are generated. Values of S_b higher than S_W can be explored through for intermediate design points between parallel implementation and smallest one with $S_b = S_W$.

5.3.6 Microarchitecture Validation Checks

When the user desires the LISA based HDL generation, the cipher configuration and selected microarchitecture undergoes following checks.

- The unroll factor, u should be a multiple of round count, $N_r = k.u$, where $k \geq 1$.
- The number of subpipelines s inserted in *round* should be the same as that of the ones inserted in *k_round*. The user may specify dummy subpipelines at the end of the *round* or *k_round* to balance the latency.
- The bitslice width S_b should be a multiple of S_W and a factor of S_B. Hence for PRESENT-80 the user gets the option of $S_b = 4, 8, 16, 32$.
- Any microarchitecture handling *multiple* data blocks cannot be designed to have non-parallel encryption or decryption mode of operation as indicated in the Table 2. For example, in OFB mode, the microarchitecture for encryption and decryption should not be subpipelined.
- Bitslicing cannot be combined with any other microarchitecture to generate a hybrid configuration.

5.3.7 RunFein Limitations

We list here some microarchitectural limitations of RunFein.

- Both software and hardware implementations generated by RunFein follow the *on-the-fly* key expansion methodology. Alternatively, subkey precomputation requires large memory for storing

$S_{SK} \times N_r$ bits of data. Additionally, the delay of subkey computation has to be incurred whenever a new key is used. RunFein does not pre-compute subkeys, however, converting the generated code to precomputed keys approach requires only trivial tweaking.
- Ciphers requiring *unequal* number of iterations for round and krounds cannot be implemented using RunFein. Though this is uncommon for most of todays ciphers, the exceptions are AES-192/256 configurations.
- For ciphers having Mix column as a diffusion operations, bitslicing requires large multiplexing logic whose overhead exceeds the potential saving achieved by bit-slicing [47]. Currently, RunFein does not support a bitsliced microarchitecture for cipher with Mix Column operation (e.g., AES). For ciphers with PBoxes, a parallel execution of PBox operation is performed instead of a bitsliced implmentation as discussed in the previous section (for PRESENT-80).
- Currently, RunFein does not support a *unified* microarchitecture performing both encryption/ decryption.

6 Experimental Results and Analysis

Using RunFein we implemented the software realizations of PRESENT $(80, 128)$, AES (128), KLEIN $(64, 80, 96)$ and LED $(64, 128)$. The software efficiency in terms of lines of code and execution time has already been discussed in [26]. The randomness test using NIST test suite was also successfully conducted by generating long streams of encrypted data in CBC, PCBC, OFB and CFB modes of operation.

6.1 Hardware Implementation and Benchmarking

We implemented various hardware microarchitectures for PRESENT-80 and AES-128. The generated high level design description models in LISA were tested with its software tools generated by *Synopsys Processor Designer* (version 2013.06-SP3) including Compiler, Assembler, Linker, Profiler and Debugger. The generated synthesizable Verilog HDL based implementations were tested for correctness using *Mentor Graphics ModelSim* (version 10.2c). All designs were synthesized with *Synopsys Design Compiler* (version G-2012.06) to have area, power and maximum frequency profiling. Design synthesis was carried out using the Faraday standard cell libraries in *topographical mode* with area optimization in mind. The area figures were converted to equivalent NAND gates (GE). We used three technology nodes for synthesis, namely, UMC L180E High Speed FSG Generic II Process 0.18μm CMOS, UMC L90 Standard Performance Low-K (Regular VT) Process 90nm CMOS and UMC SP/RVT Low-K process 65nm CMOS. The foundry typical values (of 1.8 Volt for the core voltage and 25°C for the temperature) were used. The power consumption is estimated by *Synopsys Primetime* (version 2009.12) based on gatelevel netlist switching activity by back annotation.

6.2 Microarchitectures for PRESENT-80

For *lightweight* block ciphers, low operating frequencies are more relevant due to their stringent power constraints, hence 100 KHz clock frequency is considered; results at 10 MHz are also reported. At 100 KHz, our RunFein generated PRESENT-80 encryption only loop folded implementation has a throughput of 200 Kbps and occupies 1649 GE for 65nm CMOS technology library as indicated by the first row of Table 4. The power and area results for the same loop folded implementation, synthesized at 10 MHz are indicated in the first row of Table 5.

For comparison with the manually optimized reported implementations, we take up the results for loop folded PRESENT-80 encryption estimates in [13] with three different CMOS technology libraries as indicated by the first column of Table 6. This implementation on 180nm reportedly consumes 1650 and 1706 gates at 100 KHz and 10 MHz, respectively. Our implementation, on a comparable technology library, consumes 1750 for both 100 KHz and 10 MHz operating frequency, making our results having 100 and 46 gates more, respectively [13]. This *area-gap* is far too small to be considered an overhead and possibly can be attributed to the difference in the vendor libraries, synthesis optimizations settings or different versions of synthesis tool.

Table 6 PRESENT-80 bitsliced encryption @ 100 KHz

	Area (GE) RunFein			Area (GE) [13]		
(S_b)	65nm	90nm	180nm	180nm	250nm	350nm
64	1649	1519	1751	1650	1594	1525
32	1462	1379	1602	-	-	-
16	1264	1203	1403	-	-	-
8	1182	1121	1313	-	-	-
4	1107	1081	1265	1075	1169	1000

6.2.1 Bitslicing

For bitslicing, we generated implementations with various possible bitslice width, i.e., $S_b = 4, 8, 16, 32$. Consequently the reduction in area, power and throughput is seen as a trend on 65nm CMOS technology library and

Table 4 PRESENT-80 encryption bitsliced implementation results for 65 nm CMOS tech. library @ 100 KHz

Bitsliced width (S_b)	Cycles /round	SBoxes used	Area (GE)			Power (uW)			Throughput (Kbps)
			Combinational	Sequential	Total	Static	Dynamic	Total	
64	1	16+1	896.25	752.50	1648.75	10.28	424.54	434.82	200.00
32	3	8	693.25	768.75	1462.00	9.84	121.85	131.69	66.67
16	5	4	488.25	775.50	1263.75	9.12	68.23	77.34	40.00
8	9	2	396.50	785.50	1182.00	8.98	33.46	42.44	22.22
4	17	1	128.75	978.50	1107.25	8.60	32.54	41.14	11.76

Table 5 PRESENT-80 encryption bitsliced implementation results for 65 nm CMOS tech. library @ 10 MHz

Bitsliced width (S_b)	Cycles /round	SBoxes used	Area (GE)			Power (uW)			Throughput (Kbps)
			Combinational	Sequential	Total	Static	Dynamic	Total	
64	1	16+1	891.00	752.50	1643.50	10.28	454.57	464.84	20.00
32	3	8	694.25	768.75	1463.00	9.84	150.09	159.92	6.67
16	5	4	486.00	777.00	1263.00	9.12	95.30	104.42	4.00
8	9	2	396.50	785.50	1182.00	8.98	54.92	63.90	2.22
4	17	1	128.75	978.50	1107.25	8.60	55.51	64.11	1.18

an operating frequency of 100 KHz in Table 4 and 10 MHz in Table 5. Fig. 10 and Fig. 11 graphically show the trade-off design points for area and power saving, respectively, against the loss in throughput for various S_b widths.

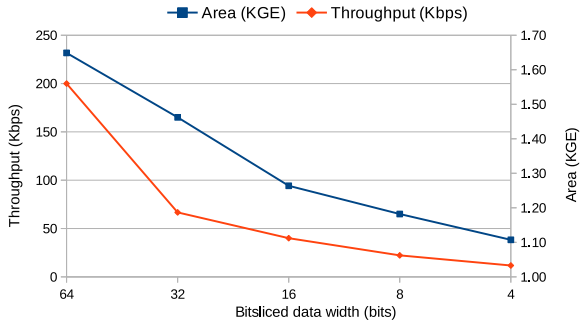

Fig. 10 PRESENT-80 bitsliced encryption area throughput trade-off @ 100 KHz

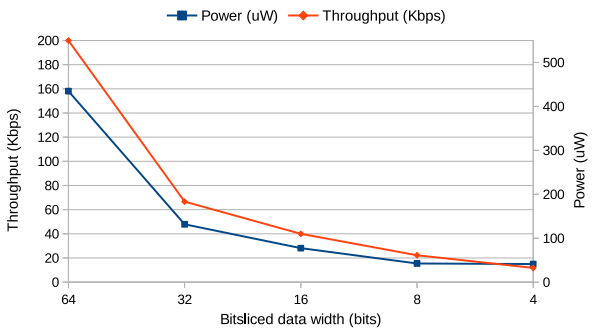

Fig. 11 PRESENT-80 bitsliced encryption area power trade-off @ 100 KHz

For comparison we take up the the smallest reported area for PRESENT-80 by *hand-crafted* implementation, requiring 1000 gates [13]. Their implementation area footprints for $S_b = 4$ on various technology libraries are reported in Table 6. For the same operating frequency (and consequently same throughput), our area estimates when synthesized on 90nm technology library come as close as 1081 GE. The implementation results for PRESENT-80 with higher bit sliced widths have not yet been reported. RunFein accelerates exploration of these intermediate design points by enabling prototyping of bitsliced architectural customizations. Some *novel* results are presented in Fig. 10 and Fig. 11 for resources-performance trade-off.

6.2.2 Unrolling without Pipelining

Using RunFein we employ various unroll factors for the 32 rounds of PRESENT-80 encryption design. Table 7 gives the area, power and throughput estimates when the design is unrolled by various factors. A fully unrolled design achieves the highest *Throughput Per Area Ratio*, however also consumes the most area and power in comparison.

Table 7 PRESENT-80 encryption unrolled implementations, 65nm @ 100 KHz (Tp is Throughtput)

Unroll factor, u	Cycles / block	Area (GE)	Power (uW)	Tp (Mbps)
1	32	1648.75	434.82	0.20
2	16	2279.75	450.00	0.40
4	8	3396.25	494.08	0.80
8	4	5859.75	522.12	1.60
16	2	10712.25	645.61	3.20
32	1	19817.75	754.00	6.40

6.2.3 Subpipelining

Through subpipelining we generated some novel high throughput realizations of PRESENT-80 cipher that

have not been reported till date. For a loop folded implementation , the maximum operating frequency is profiled to be 3.7 GHz as indicated by the Table 8. We subpipeline it *twice* for achieving high throughput performance.

- *First Subpipeline*: The critical path for the loop folded implementation (Fig. 5 left) exists from the *k_state* register, through the 3 round layers, the multiplexer and till the *d_state* register. Since PBox poses no combinational delay due to rewiring, its prudent to break the critical path by inserting a subpipeline between *layer*0 and *layer*1 of the cipher, shown by the single dotted line in Fig. 5. A corresponding subpipeline between *layer*1 and *layer*2 of the *k_round* is also opted. Consequently the subpipelined circuit's operating frequency increases, raising the throughput to 8.1 Gbps.
- *Second Subpipeline*: The critical path now exists between the *subpipeline register* and the *d_state* register in round. For a further increase in the operating frequency we break this critical path between *layer*1 and *layer*2 of *round* by a second subpipeline (with a corresponding subpipeline between *layer*0 and *layer*1 of *k_round*) as shown by double dotted lines in Fig. 5. The corresponding operating frequency however *decreases*. This is attributed to the supporting control hardware inserted to tackle the 2 subpipelines. A 2-bit supplementary counter (*s_counter*) counting up to the number of subpipelines is inserted in addition to the 5-bit *counter* for rounds. The critical path now exists in the controller, i.e., between *s_counter* and *counter*, prohibiting further speedup by pipelining.

Table 8 PRESENT-80 subpipelined encryption results for 65nm CMOS, (Tp is Throughtput)

s	Max Freq. (GHz)	Area (GE)			Tp (Gbps)
		comb.	seq.	total	
0	3.71	2502.75	856.00	3358.75	7.42
1	4.05	2320.75	1803.50	4124.25	8.1
2	4	2818.75	2657.00	5475.75	8.0

6.3 Microarchitectures for AES-128

For comparison of RunFein generated realization for AES-128 with a similar architecture hand-crafted realization, we took up the RTL implementation of a loop folded AES-128 encryption core available at Open Cores [37]. Since RunFein does not register the I/Os of the cipher implementation, we removed the registers for plaintext and ciphertext from open cores RTL for enabling equitable comparisons. Both of these RTL realizations were synthesized using the 65nm technology library with *same* versions of synthesis tools and settings at 10 MHz and 100 MHz operating frequencies, the area footprints obtained are comparable as shown in Table 10.

The area overhead of around 5% for the opencores RTL is attributed to its several differences compared to RunFein design. *Firstly*, instead of putting a multiplexer for bypassing the GF-mul stage in AES round, a separate layer of 128 bit XORs is inserted to get the ciphertext after the last round. *Secondly*, it maintains a 32-bit register to retain RCON value from a LUT, RunFein has no register for that. The consequent sequential area overhead can be seen in Table 10. *Thirdly*, it does not reuse the 32 bit XORs for calculation of keywords in *layer*3 till *layer*6 of key rounds. Consequently, 5 XORs (32 bits each) are used for least significant keyword, 4 XORs for the words next to it and so on. RunFein uses only 5 XORs in total for that, consequently their area overhead for combinational logic is higher.

Table 10 AES-128 encryption results for 65nm CMOS

Source	Op. freq. (MHz)	Area (GE)		
		Comb.	Seq.	Total
Opencores [37]	10	14540	1389	15929
	100	14600	1389	15989
RunFein	10	13825	1300	15125
	100	13867	1300	15167

6.3.1 Unrolling without Pipelining

The loop based AES-128 implementation may be unrolled by a factor of 2, 5 or 10 for a potential increase in the throughput performance of the design. Table 9 gives the increase in area and consequently the throughput improvement when the design is unrolled and profiled for the maximum achievable frequency. Interestingly, the highest throughput/ area efficiency of the design is achieved with unroll factor 2. For higher values of loop unrolling, the gain in throughput is diminished by the large number of SBoxes and wide bus based selection circuitry.

6.3.2 Subpipelining

For a loop folded generated implementation of AES-128, the maximum operating frequency is profiled to be 1.65 GHz as indicated by the Table 11. The critical path is found to exist from the *d_state* register, through the 4 round layers, the multiplexer and back to the *d_state* register. To break this critical path we indicate RunFein to place a subpipeline between *layer*0

Table 9 AES-128 unrolled encryption implementation results for 65nm CMOS tech. library

Unroll factor (u)	no. of SBoxes	Max. Freq (GHz)	Area (GE)			Throughput (Gbps)	Throughput/Area (Mbps/GE)
			Combinational	Sequential	Total		
1	16+4	1.65	54666.00	1461.50	56127.50	21.12	14.45
2	32+8	0.90	120293.50	1449.75	121743.25	23.04	15.89
5	80+20	0.30	169780.25	1406.75	171187.00	19.2	13.65
10	160+40	0.12	704315.25	1474.25	705789.50	15.36	10.42

and *layer*1 of the cipher *round* and a corresponding pipeline between *layer*1 and *layer*2 of the *k_round*, as shown by the single dotted line in Fig. 5. The RTL for the pipelined architecture is profiled to operate on a frequency as high as 2.25 GHz, with a 28.8 Gbps of throughput. The critical path now exists between *d_state* register and the pipeline register, i.e., the SBox layer. A further exploration of breaking critical path is possible by partitioning the SBox tables into 2 or more levels (instead of using one 256 entry SBox we use 8 with 32 entry SBoxes) and inserting pipelining in between. Similarly, the Galois field inversion of the S-box using sub-fields of 4, 2 bits can be used for lower area footprints. The required multiple layers of operations for sub-fields inversion and operations can be subpipelined for achieving higher performance [46].

Table 11 AES-128 subpipelined encryption results for 65nm CMOS

s	max freq. (GHz)	Area (GE)			Throughput (Gbps)
		comb.	seq.	total	
0	1.65	54666	1461.50	56127.50	21.12
1	2.25	49896	3464.75	53360.75	28.8

7 Conclusion and Future Work

We present RunFein, an extensible framework for the rapid prototyping of block ciphers into customizable hardware and software implementations. It offers a sophisticated design capture of the algorithmic and structural specifications of a cipher by the user through a GUI. The algorithmic design requires specification of layers of atomic operations for key expansion and round transformations. The hardware implementation is aided by a commercial high-level synthesis framework. The architectural specifications of a loop folded configuration of cipher is automatically transformed by RunFein according to the microarchitecture configuration specified by the user (loop unrolling, bitslicing, subpipelining). A thorough design viability is validation before design rapid prototyping. We took up some noticeable block ciphers with various different architectural specifications for implementation using RunFein. Equitable comparisons for area-throughput-power were carried out. Our results rivals the best available handwritten IP cores. Additionally, some novel optimization's results for PRESENT-80 (bitslicing) have also been reported.

RunFein's high-level design approach eliminates the laborious development efforts for VLSI realization/ verification of block ciphers. It aids the cryptographic community by enabling speedy benchmarking against critical resources like area, throughput, power, latency and allows allows design exploration of various microarchitectural design alternatives. We see RunFein as a first instance of a tools framework suite for high-level realization of domain-specific cryptography functions (block ciphers). Extensions to other cryptograhpic functions would follow. We are enthusiastic to extend this work in various directions.

- A similar rapid prototyping tool for stream ciphers, called *RunStream*, is in the pipeline.
- Inclusion of cryptanalytic tools for block ciphers ciphers is intended.
- An automatic software generation of parallel programming for GPU-accelerated machines is on the roadmap.
- We plan to take up *unified* hardware microarchitecture supporting both encryption/ decryption of ciphers.

References

1. A. Rukhin, J. Soto, J. Nechvatal, M. Smid and E. Barker. A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications, NIST Special Publication 800-22 Available at `csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22b.pdf`.
2. Advanced encryption standard. Federal Information Processing Standard, FIPS-197 (2001): 12.
3. Secure Hash Standard (SHS) In FIPS PUB 180-4, Information Technology Laboratory National Institute of Standards and Technology Gaithersburg, March 2012 Available at `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`.
4. D. J. Wheeler, R. M. Needham. TEA, a tiny encryption algorithm. In Fast Software Encryption, Springer Berlin Heidelberg, pp. 363-366, January 1995
5. H. Wu. The stream cipher HC-128. Available at `http://www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf`.

6. L. Batina, J. Lano, N. Mentens, S. B. Ors, B. Preneel and I. Verbauwhede. "Energy, performance, area versus security trade-offs for stream ciphers. In The State of the Art of Stream Ciphers" In The State of the Art of Stream Ciphers, ECRYPT Workshop Record, pp. 302-310, 2004
7. A. Chattopadhyay and G. Paul. "Exploring security-performance trade-offs during hardware accelerator design of stream cipher RC4." In VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on. IEEE, 2012.
8. eSTREAM: the ECRYPT Stream Cipher Project. Available at `http://www.ecrypt.eu.org/stream`.
9. F. Chabaud and A. Joux. "Differential collisions in SHA-0." In Advances in CryptologyCRYPTO'98, Springer Berlin Heidelberg. pp. 56-71,1998
10. Break DES in less than a single day In Press release demonstrated at a 2009 workshop Available at `http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html`.
11. S. Maitra and G. Paul. "Analysis of RC4 and proposal of additional layers for better security margin." In Progress in Cryptology-INDOCRYPT, pp. 27-39. Springer Berlin Heidelberg, 2008.
12. A. Bogdanov, L. R. Knudsen, G. Le, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Proceedings of CHES 2007.
13. Rolfes, C., Poschmann, A., Leander, G., and Paar, C. Ultra-lightweight implementations for smart devicessecurity for 1000 gate equivalents. In Smart Card Research and Advanced Applications, Springer Berlin Heidelberg, pp. 89–103, 2008.
14. J.Aumasson, L. Henzen, W. Meier and R. Phan. SHA-3 proposal BLAKE ver 1.3, 2010. Available at `https://www.131002.net/blake`.
15. D. J. Bernstein. The Salsa20 family of stream ciphers. In New Stream Cipher Designs: The eSTREAM Finalists, Springer-Verlag, 2008, pp. 84–97.
16. M. Dworkin. Recommendation for block cipher modes of operation. Methods and techniques. In NIST Special Publication 800-38A, 2001
17. C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin and H. Sibert Sosemanuk, a fast software-oriented stream cipher. In New Stream Cipher Designs: The eSTREAM Finalists, Springer-Verlag, 2008, pp. 98–118.
18. N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas and J. Walker. The Skein Hash Function Family, Version 1.3. `http://www.skein-hash.info/sites/default/files/skein1.3.pdf`, October 2010.
19. Barreto, P. and V. Rijmen,. The Whirlpool hashing function. In First open NESSIE Workshop, Leuven, Belgium, Vol. 13, pp. 14–33, 2000
20. Authenticated encryption-security techniques In ISO/IEC 19772:2009. Retrieved March 12, 2013.
21. R.-P. Weinmann. AXR - Crypto Made from Modular Additions, XORs. In Dagstuhl Seminar 09031, January 2009. Available at `http://www.dagstuhl.de/Materials/Files/09/09031/09031.WeinmannRalfPhilipp.Slides.pdf`.
22. G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. Keccak sponge function family main document. Submission to NIST, round 3, 2011.
23. R. Rivest. The MD5 Message-Digest Algorithm. In RFC 1321 by MIT Laboratory for Computer Science and RSA Data Security, April 1992 Available at `http://www.faqs.org/rfcs/rfc1321.html`.
24. D. Khovratovich and I. Nikolić. Rotational cryptanalysis of ARX. In Fast Software Encryption 2010, LNCS vol. 6147, Springer, pages 333–346.
25. T. Iwata, K. Shibutani, T. Shirai, S. Moriai and T. Akishita. AURORA: A Cryptographic Hash Algorithm Family. Submission to NIST, 2008.
26. A. Khalid, M. Hassan, A. Chattopadhyay and G. Paul. RAPID-FeinSPN: A Rapid Prototyping Framework for Feistel and SPN-Based Block Ciphers. In Information Systems Security (pp. 169-190). Springer Berlin Heidelberg. 2013
27. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S.W. Williams and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. UCB/EECS-2006-183, EECS Department, University of California, Berkeley
28. P. Dubey. Teraflops for the masses: Killer apps of tomorrow. In Workshop on Edge Computing Using New Commodity Architectures (UNC), 2006, volume 23.
29. G. Leurent. ARX Toolkit. Available at `http://www.di.ens.fr/~leurent/arxtools.html`.
30. N. Mouha, V. Velichkov, C. De Canniére and B. Preneel. S-function Toolkit. Available at `http://www.ecrypt.eu.org/tools/s-function-toolkit`.
31. M. Ernst, S. Klupsch, O. Hauck and S. A. Huss. Rapid Prototyping for Hardware Accelerated Elliptic Curve Public-Key Cryptosystems. In Proceedings of the 12th International Workshop on Rapid System Prototyping (RSP '01), 2001.
32. Akinyele, Joseph A., et al. Charm: A framework for rapidly prototyping cryptosystems. In Journal of Cryptographic Engineering, pp. 1-18, 2013
33. Lacy, John B., Donald P. Mitchell, and William M. Schell. CryptoLib: Cryptography in software. In Proc. of Fourth USENIX Security Workshop, pp. 1-18, 1993.
34. K. Shahzad, A. Khalid, Z. E. Rákossy, G. Paul and A. Chattopadhyay. CoARX: a coprocessor for ARX-based cryptographic algorithms. In Proceedings of the 50th Annual Design Automation Conference (DAC '13), doi=10.1145/2463209.2488898, 2013.
35. A. Chattopadhyay, H. Meyr and R. Leupers. LISA: A Uniform ADL for Embedded Processor Modelling, Implementation and Software Toolsuite Generation. In *P. Mishra, N. Dutt (editors) Processor Description Languages, Morgan Kaufmann*, pp. 95–130, 2008.
36. Synopsys Processor Designer. Available at `http://http://www.synopsys.com/Systems/BlockDesign/processorDev/Pages/default.aspx`.
37. Simple AES (Rijndael) IP Core `http://opencores.org/project,aes_core`.
38. Announcing development of a federal information processing standard for advanced encryption standard. National Institute of Standards and Technology, Docket No. 960924272-6272-01, RIN 0693-ZA13, January 2, 1997. `http://csrc.nist.gov/archive/aes/pre-round1/aes_9701.txt`
39. NESSIE: New European Schemes for Signatures, Integrity, and Encryption IST-1999-12324, January, 2000. `https://www.cosic.esat.kuleuven.be/nessie/`
40. CRYPTREC: Cryptography Research and Evaluation Committees. Japanese Government Cryptographer Competition. March 7, 2012. `http://competitions.cr.yp.to/cryptrec.html`
41. SHA-3 Cryptographic Hash Algorithm Competition. NIST competition for Secure Hash Algorithm, 2007. `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`

42. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness A portfolio of authenticated ciphers, 2013. http://competitions.cr.yp.to/caesar.html
43. Third Round Report of the SHA-3 Cryptographic Hash Algorithm Competition. National Institute of Standards and Technology, NISTIR 7896, November 2012. Available at http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf
44. Gaj, Kris, Jens-Peter Kaps, Venkata Amirineni, Marcin Rogawski, Ekawat Homsirikamol, and Benjamin Y. Brewster. "ATHENa-automated tool for hardware evaluatioN: toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs." In Field Programmable Logic and Applications (FPL), 2010 International Conference on, pp. 414-421. IEEE, 2010.
45. Menezes, Alfred J., Paul C. Van Oorschot, and Scott A. Vanstone. "Handbook of applied cryptography." CRC press, 1996.
46. Satoh, Akashi, et al. "A compact Rijndael hardware architecture with S-box optimization." Advances in CryptologyASIACRYPT 2001. Springer Berlin Heidelberg, 2001. 239-254.
47. Moradi, Amir, et al. "Pushing the limits: A very compact and a threshold implementation of AES." Advances in CryptologyEUROCRYPT 2011. Springer Berlin Heidelberg, 2011. 69-88.

Appendix

We present here some GUI snapshots of various tabs of RunFein tool. CRYKET (CRYptographic Kernels Toolkit) caters to rapid prototyping of various cryptographic functions while RunFein is an instance of it dealing with block ciphers.

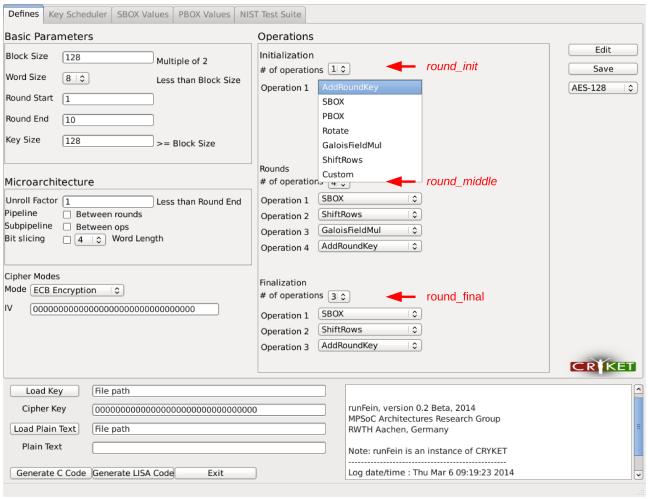

Fig. 12 Round layers operational specification for AES-128 in RunFein

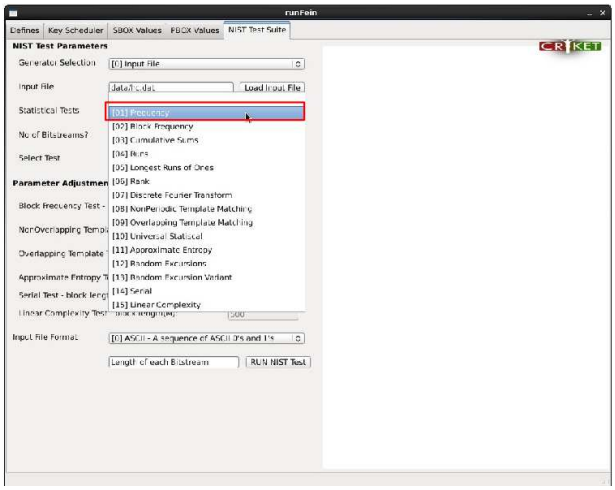

Fig. 13 NIST Test Suite parameter selection tab in RunFein

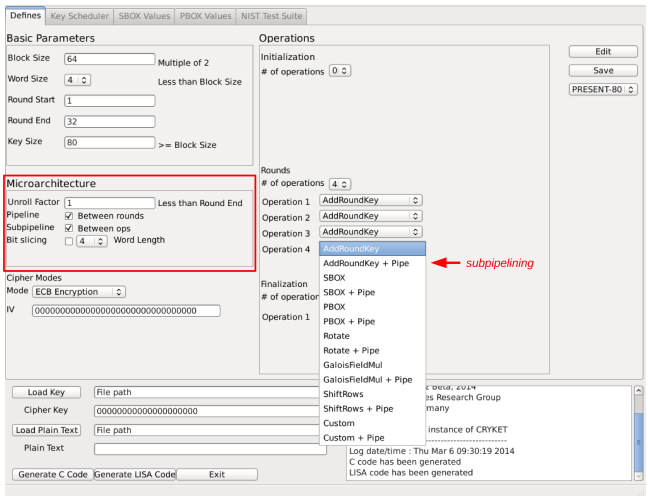

Fig. 14 Microarchitectural specification for subpipelined implementation in RunFein (+ *pipe* specifies pipeline)